How Do Software Architects Specify and Validate Quality Requirements?

Andrea Caracciolo, Mircea Filip Lungu, and Oscar Nierstrasz

Software Composition Group, University of Bern, 3012 Bern, Switzerland
{caracciolo,lungu,oscar}@iam.unibe.ch
http://scg.unibe.ch

Abstract. Software architecture is the result of a design effort aimed at ensuring a certain set of quality attributes. As we show, quality requirements are commonly specified in practice but are rarely validated using automated techniques. In this paper we analyze and classify commonly specified quality requirements after interviewing professionals and running a survey. We report on tools used to validate those requirements and comment on the obstacles encountered by practitioners when performing such activity (*e.g.*, insufficient tool-support; poor understanding of user's needs). Finally we discuss opportunities for increasing the adoption of automated tools based on the information we collected during our study (*e.g.*, using a business-readable notation for expressing quality requirements; increasing awareness by monitoring non-functional aspects of a system).

Keywords: software architecture, empirical study, quality requirements, validation

1 Introduction

The primary task of a software architect is to define and specify a suitable high-level design solution that fulfills all major technical and operational requirements. The document describing the architecture provides requirements and guidelines that will help in maintaining the conceptual and technical integrity of a software product. Quality requirements describe expected characteristics of specific aspects of the system, from its implementation to its observable behavior. They may refer to externally visible product qualities (*e.g.*, performance requirements) or to implementation details that support them (*e.g.*, legitimate module dependencies). Ensuring the enforcement of quality requirements and their deriving constraints should prevent architectural decay and make the system more adaptable to new, emerging requirements [3].

In this study we set out to survey whether the definition of quality requirements is a common practice in IT companies. We want to understand whether this activity is systematic and supported by tools and processes or rather based on personal assumptions and using makeshift tools. Finally, we are interested whether quality requirements, given their importance, are also automatically validated as the software system evolves.

Previous studies [7,8,13] propose solutions for specifying architectural invariants. Other studies [1,18,19,10] rank non-functional qualities (*e.g.*, performance, usability, availability, *etc.*) by carrying out surveys. In neither case is effort made to explore quality attributes from the point of view of practitioners.

In our study we focus on the following research questions:

1. What kind of quality requirements do architects define in practice?
2. How are quality requirements specified?
3. How are quality requirements validated?

To answer these questions, we use empirical methods to identify quality attributes that practitioners consider when designing their architecture. Furthermore we analyze how practitioners specify quality requirements in their documentation and explore the various techniques that are used for validation.

We observe that architects do not always adopt automated techniques to validate quality requirements and when they do, they automatically verify only a small subset of all the specified requirements. We discuss possible obstacles that might cause this situation as well as research opportunities that could lead to a general improvement in the practice of quality requirements validation (Section 5). Among the identified opportunities we consider the advantages of adopting a business-readable declarative language for specifying quality requirements. We also explore the benefits of promoting architectural visibility by introducing continuous validation support for user-defined quality requirements in current monitoring platforms (*e.g.*, Sonarqube).

2 Research Method

This paper uses a mixed research methods strategy: *sequential exploratory design* [4]. This approach consists of two different research methodologies: a qualitative investigation followed by a quantitative validation survey which triangulates the results of the first.

In the first study, we focused on collecting qualitative data. The goal of this study was to gain a possibly comprehensive overview of the state of practice in the definition and validation of quality requirements. The questions have been iteratively refined by conducting three internal pilot interviews with PhD and master students with professional experience in the field. The final list of questions, used as loose guideline for the actual interviews, is available on our web site[1]. Fourteen people working for six different organizations agreed to participate in our study (Table 1). More than 70% of the participants have been contacted indirectly through an intermediary and had no relevant links to the academic community. The remaining subjects were contacted directly and belonged to our industrial collaboration network. All interviews were carried out independently, leading to a set of complementary and partially overlapping observations. A total of approximately 18 hours of conversation have been recorded.

[1] `http://scg.unibe.ch/research/arch-constr/study`

#	Role	Org.	Project (domain; type)	team size
A	CEO, architect	C1	government / enterprise	<5
B	business manager	C2	government / enterprise	10-50
C	project manager	C3	insurance / enterprise	>50
D	architect	C4	logistic / enterprise(integration)	<5
E	developer	C4	logistic / enterprise(integration)	<5
F	CTO	C5	banking / enterprise	>50
G	architect	C2	government / enterprise	5-10
H	architect	C2	government / enterprise	10-50
I	architect	C6	logistic / enterprise(migration)	>50
J*	developer	C2	government / development support tool	<5
K	architect	C5	banking / enterprise	5-10
L	architect	C6	transportation / control systems	5-10
M*	developer	C5	banking / source code analysis	>5
N*	architect	C5	banking / development support tool	5-10

Table 1. Interview study participants. Candidates with an asterisk worked in projects aimed at supporting architectural design. The remaining candidates worked as software architects or project managers in medium to large projects and have more direct experience in architectural design.

The main outcome of this qualitative study was the list of quality attributes presented in Table 2. These quality attributes were inferred by analyzing the interviews and synthesizing the main concerns using coding techniques [17]. To support this activity, we identified and labeled quality requirements in interview transcriptions as well as the documentation files (*i.e.*, Software Architecture Documents, Developer guidelines) that we collected at the end of several interview sessions. To gather more evidence that the observations coming from the first study actually reflected the state-of-practice of a broader community, we created an e-survey. Over a time span of two months we collected 34 valid and complete responses. Invitations were sent to professionals selected among industrial partners and collaborators (*i.e.*, convenience sampling method), including people involved in the first phase of the study. The survey was also advertised in several groups of interest related to software architecture hosted by LinkedIn and on Twitter[2] (*i.e.*, voluntary sampling method). Survey participants were asked to specify whether the quality attributes identified in the first study were ever encountered in a past project, their perceived level of importance (on a scale from 1 to 5, with 5 being the highest), the formalism adopted to describe them and the testing tool used for their validation. A complete copy of the survey can be found on our web site[1].

3 Learning from Practitioners: a Qualitative Study

During interviews, we tried to elicit a possibly wide range of distinct architecturally significant quality attributes. We asked our respondents to enumerate

[2] `http://www.linkedin.com`; `http://www.twitter.com`

those concerns that could be considered fundamental for their architecture. For each of those, we asked them to describe their main properties and the form in which they were typically specified. Table 2 shows all identified quality attributes. For each quality attribute, we also present additional details collected during our quantitative study (columns 3-6 in Table 2).

Quality attributes are categorized based on the closest matching ISO-25010[11] quality characteristic. For simplicity's sake, we decided to pair each attribute with one single category. For clarity, we also published some explanatory requirements for all presented quality attributes on our web site[1].

Quality Characteristic	**Quality Attribute** (**I**nternal / **E**xternal / **P**rocess)	**Importance** Q_1	Q_2	Q_3	**Fam.**	**Form. Not.**
Performance	**Response time** (E)	3	4	5	15%	14%
	Throughput (E)	3	4	4	26%	13%
	Hardware infrastructure (I)	2	3	4	29%	0%
Compatibility	**Signature** (I)	3	4	4	18%	52%
	File location (I)	1	3	4	29%	18%
	Data structure (I)	2	3	4	29%	47%
	Communication (I)	2	4	4	15%	22%
Usability	**Visual design** (E)	2	3	3.5	9%	21%
	Accessibility (E)	1	2	3.5	50%	0%
Reliability	**Availability** (E)	4	4	5	15%	14%
	Recoverability (E)	2	3	5	32%	5%
	Data integrity (I)	3	3	4	18%	23%
	Event handling (I)	2	3	4	35%	25%
	Software update (P)	1	2	3	59%	0%
Security	**Authorization** (E)	4	4	5	3%	23%
	Authentication (E)	3	4	5	21%	12%
	Data retention policy (I)	2	3	4	12%	13%
Maintainability	**Meta-annotations** (I)	1	3	4	32%	39%
	Code quality (I)	2	3	3.5	15%	19%
	Dependencies (I)	2.5	3	4	18%	53%
	Naming conventions (I)	2	3	3	12%	38%
Portability	**Software infrastructure** (I)	3	3	4	24%	8%

Table 2. Taxonomy of quality requirements (grouped by supported quality characteristic). Columns (from left to right): Matching quality characteristic; Quality requirement; Evaluated importance (first, second and third quartile); Participants who encountered the requirement in a previous project (familiarity); Participants who specified the requirement using a formal notation. Columns 3-6 contain data collected during our quantitive study.

3.1 Identified Quality Attributes

We now comment on the identified quality attributes.

Performance: performance was often mentioned as being a key concern. Requirements on *response time* and *throughput* are commonly part of the acceptance criteria defined with the customer at the beginning of a project. Several respondents (*e.g.*, **A**, **B**) define latency requirements on the execution of specific tasks (*e.g.*, The system has to answer each request within 10 ms). Others (*e.g.*, **A**, **D**) set limits for the accepted throughput (*e.g.*, The system must be able to execute a certain task 10'000 times per hour). These requirements are often validated by collecting timestamps during execution or simulating high traffic load with a script. *Hardware infrastructure* requirements, specifying the hardware resources required to support a specific software implementation, also play a role in determining performance.

Compatibility: multiple interviewees (**B, F, J**) mentioned *communication* as one of the most important aspects in their architecture. **F** built a client simulator to test conformance with the prescribed communication protocol and check syntactical/semantical data consistency. **N** defined a guideline stating that data has to be passed from one layer to the other using Data Transfer Objects. **G** wrote a detailed specification of all service interfaces composing his application (*signature* attribute). This included details regarding accepted parameter values and activity diagrams describing the message exchange protocol. Interoperability between different components and tools often requires files to be placed into pre-determined folders or structure files according to a given shared schema (*file location* attribute).

Usability: *visual design* and compliance to *accessibility* guidelines were mentioned as typical requirements for application front-ends. **H** developed a web interface that had to conform to a set of rules defined in the corporate visual style guide. This requirement was satisfied by defining global stylesheets and forcing their inclusion into all related applications.

Reliability: robustness and fault-tolerance are important features for almost any kind of application. **H**'s application was required to guarantee 96% *availability* and a clear recovery procedure was defined for each type of fault that was likely to occur. *Data integrity* is also a major concern. **K** managed to maintain internal data consistency by defining data type classes for all supported business value types. **H** and **G** constrained field values specifying Hibernate or Spring formatting annotations. Specific rules were also defined to regulate strategies for *handling events* (*e.g.*, exceptions, notifications) and *update software* packages (*e.g.*, libraries).

Security: security is also considered critical and is often tested thoroughly. Verification becomes a necessity when the system is directly exposed to a large untrusted audience. Testing seems to have lower priority if the application is just deployed within an intranet (**E**). Most of the time, widely known frameworks (*e.g.*, JAAS) are used to implement *authentication* and *authorization* rules.

Maintainability: class *dependencies* and syntactic code invariants are commonly considered tightly related to software architecture. **H** even claims that "dependencies between modules are the main characteristic of a software architecture". Requirements on these two aspects are defined to support architectural principles (loose coupling, high cohesion) and minimize the cost of future maintenance.

Portability: requirements related to *software infrastructure* configuration are common. Prescriptions on technologies to be adopted can be found in almost every specification document. **J**, for example, specifies that the "persistence layer" of his application must use Hibernate as a persistence framework. Software infrastructure requirements are often related to rules addressing compatibility issues (*i.e.*, file location, data structure).

3.2 Specifying Quality Requirements

All the participants of our study describe their quality requirements in one or more text documents. The vast majority adopt a well-known standard template (*e.g.*, 4+1[12], togaf[3], arc42[4]). Textual documentation is always complemented with diagrams based on a common shared visual language (*e.g.*, UML, BPML, BPEL, flowchart, informal notation).

Documentation Audience Documentation is written to satisfy the needs of three main stakeholders: customers, architects and developers.

For customers: documentation is written to meet contractual requirements. In this case documentation is often seen as a burden for the architect and provides limited support to practitioners working on the project. It provides a non-technical specification that can be used to prove compliance to agreed requirements during a post-development validation phase (**G**).

For architects: documentation is written to maintain a general overview of the system and support high-level design reasoning. Some respondents believe that developers are not interested in reading about architecture. "Developers only care about functionality and tend to ignore non-functional properties" (**E**). This assumption supports the idea that architecture and implementation are on different levels of abstraction and are hard to link together. Low effort is usually dedicated to keep documentation aligned and up-to-date with changes originated in the implementation. **I** stated that he rarely got any sort of feedback from the assumed recipients of his documentation work.

For developers: documentation is a map, providing a high-level description of the system to technical users involved in the development process. It is particularly useful as an initial entry-point for new developers learning about the system. **D** said that "new developers start by reading the documentation, look into the code and finally sort out remaining doubts by talking with colleagues". Documentation is used to transfer knowledge, is open for change and needs to be kept up-to-date.

Documentation Intent In our study we identified two type of documentation styles: descriptive and prescriptive.

[3]`http://www.opengroup.org/togaf`

[4]`http://www.arc42.de`

Descriptive documentation: is meant to provide sufficient evidence to support developers in decision making activities. It is not written to set precise guidelines and rules but to help developers in evaluating alternatives and make good design choices. Architects writing "descriptive documentation" are usually skeptical about enforcing design rules through documentation. **D** said that "documented rules are often perceived as pedantic and restrictive". He added that "forcing developers to learn them beforehand is a failing strategy and often leads to poor results" because "they could be ignored and neglected". Apparently a much better approach is to provide useful feedback to developers when they break such rules.

Prescriptive documentation: is more oriented towards the definition of strict guidelines and rules. The goal is to limit developers in their design choices in order to guarantee high-level properties (*e.g.*, maintainability). In this case, it's often convenient to express quality requirements in a clear and objective way. Most of the documents collected during our studies contained coding guidelines (general practices and syntax format rules) and quality requirements regarding data values and event handling.

Formalization of Quality Requirements Quality requirements are rarely described formally. Formal specification is only used in practice to support specific verification tools. In this case, users are forced to extract architectural rules from the specification document and encode them in a separate file using a tool-specific notation.

In rare cases, companies develop their proprietary description language. **N** worked in a company where all developed applications are documented as visual diagrams based on a proprietary meta-model. Their models include a hierarchically organized set of interlinked logical components. All types of entities are characterized by various properties (*e.g.*, interface structure for components; message format, protocol, integration type for communication links). Each system, consisting of a set of components, is mapped to the specific infrastructural entity on which it is supposed to be deployed. This last information is used to feed a semi-automatic process for verifying the actual deployment configuration. **N** said that the documentation model adopted in his company is very helpful for keeping information consistent, accurate and closed to interpretation.

In other cases, users face the lack of usability of current specification mechanisms. **D**, for example, decided to verify package dependencies using a specific testing framework (JDepend). Unfortunately the test specification required by the adopted tool was not readable enough to be included in the official documentation. To solve this problem, he decided to specify the requirements in a spreadsheet and build a parser to generate a corresponding set of tests. In this case, having a simplified and testable representation of architectural rules justified the cost for building a conversion tool.

3.3 Validating Quality Requirements

We observed that quality requirements are validated using various approaches.

Manual Validation According to the answers collected during our study, one way of validating quality requirements is simply by running the system and manually checking some operational properties (*e.g.*, *Response time*, *Authentication*). This validation strategy is usually preferred when automated testing tools are not available or exist but are too expensive to buy or customize. Scalability is sometimes verified by generating a large number of requests using a script and evaluating responsiveness by interacting with the application through an additional session. Properties that manifest themselves in source code (*e.g.*, *Code conventions*), are often checked through code reviews. As mentioned by **L**, "the number of existing [testing] tools is far from being exhaustive". He said that "companies rarely see the value of investing time in researching new testing techniques". In many cases manual validation seems to be the most viable and frequently chosen alternative.

No Validation Some respondents avoid the need for direct verification by relying on a framework or code generator. If the framework is not developed internally, the fact that certain quality requirements are actually fulfilled is based on trust. **J**, responsible for the development of an internal framework used across multiple company projects, said that "frameworks should not be invasive but support the developer by simplifying his tasks and reducing possible design decisions in a non-invasive way". Frameworks that are built to limit implementation choices, as confirmed by **M**, are not well perceived by developers. A framework should convince developers to use its functions by offering useful services that contribute to reducing the cost of development (**J**). Code generators are typically used to simplify the maintenance and creation of modules that depend on business needs that vary through time. Our interviewees agreed on the fact that building testing tools is usually not an economically viable option. Building testing tools is also seen as a challenging task requiring advanced programming skills.

Automated Validation When possible, architects prefer to use automated techniques. This can be done by writing programmatic tests or relying on tools developed by a third party. Existing tools do not always fit the needs of our respondents. Multiple respondents said that some of the currently available tools were lacking in flexibility and usability. **F** worked on a project where components could be identified by looking at the suffix of class names. All the tools he tried supported package name matching as the only mapping strategy. **K** was working on a system based on the OSGi framework[5]. He was not aware of any tool that allowed him to automatically check whether the specified dependencies existing between the OSGi bundles composing his system were actually consistent with the architectural specification. The only way to verify the alignment between implementation and specification was to manually inspect large XML configuration files.

[5] `http://www.osgi.org/`

Most of the tools force users to operate on an overly technical level. This fact prevents non-technical stakeholders from accessing valuable information and introduces new costs for setting-up and maintaining architectural tests. Current testing solutions require the user to specify testing rules in separate files. Quality requirements must be specified twice: in the official documentation using natural language (for supporting communication and reasoning) and in a purpose-built formal specification file (for supporting a specific testing solution). The resulting fragmentation leads to increased costs for maintaining multiple specifications aligned and consistent.

4 Corroborating the Evidence: a Quantitative Study

To confirm the validity of our impressions on a larger scale, we developed a second study. This study was aimed at obtaining a more uniform overview on how quality attributes (identified in the first study and presented in Table 2) are considered by practitioners.

We now report some of the main observations resulting from the analysis of the obtained results.

O1. Most requirements are not formally specified: Our survey confirms that very few requirements are formally or semi-formally specified (Table 2). In fact, only 2 quality requirements (*Signature*, *Dependencies*) out of 22 are formally specified more than 50% of the time. *Signature* quality requirements are specified using UML with custom profiles, XSD and IDLs (OMG IDL, MIDL, WSDL). *Dependencies* are described using tool-specific notations (*e.g.*, JDepend, ndepend, macker, DCL, SOUL), Java annotations and UML with custom profiles. Others (*Data structure*, *Naming conventions*) are also quite frequently formalized. *Naming conventions* can be specified using regular expressions, EBNF grammars, tool-specific notations (*e.g.*, SOUL for IntensiVE) or Java (*e.g.*, plugins for Checkstyle and PMD). *Data structure* quality requirements are either specified using standard schema definition languages (DTD, XSD) or semi-formal modeling notations (ER, UML).

O2. Automated testing is not commonplace: Results show that the use of automated techniques (*i.e.*, using white-/black-box testing or tools) for validating quality requirements is not commonplace (Figure 1). On average, 59% of the surveyed population adopts non-automated techniques (*e.g.*, code review or manual validation) or avoids validation completely. Based on the results of our survey (Figure 1), the following quality requirements are mostly validated manually: *Dependencies* (10 users), *Visual design* (8), *Naming conventions* (7), *Communication* (5). Quality requirements that remain most often unvalidated are: *hardware infrastructure* (50% of respondents), *recoverability* (48%) and *software update* (44%). Automated validation is not commonplace and is mostly adopted to validate quality requirements regarding end-user properties (*e.g.*, *Response time*, *Throughput*) and security (*e.g.*, *Authorization*, *Authentication*, *Data retention policy*). Table 3 shows which tools are used by the participants of our survey

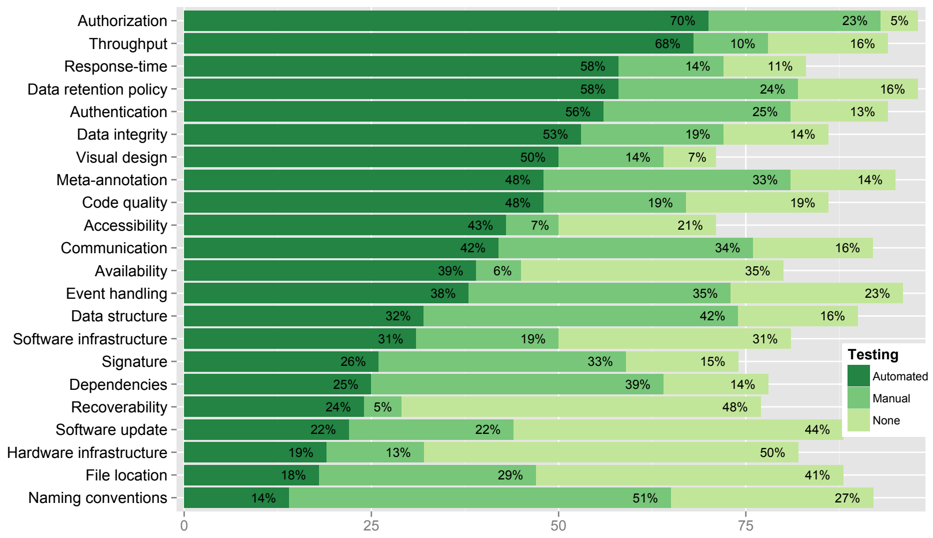

Fig. 1. Survey results: various approaches for validating quality requirements.

to validate the identified quality attributes.

O3. Tool support for automated validation is insufficient: One of the reasons why automated validation is not widespread seems to be related to the scarce availability of industrial-strength tools matching some practitioner's needs. A number of quality attributes (*e.g.*, *Code dependencies*, *Naming conventions*) can be checked with a large number of tools, while others (*e.g.*, *Data integrity*, *Meta-annotations*), considered as equally important, can only rely on a much smaller range of solutions.

O4. User's needs are still not completely recognized: Figure 1 shows that several requirements are also more frequently validated manually than automatically. The most striking examples are *Data structure*, *signature*, *dependencies*. This suggests the possibility that some requirements are still left unaddressed and need to be investigated further by conducting on-the-field studies. We believe that further analysis of emerging requirements could lead to new opportunities for future research in the field of tool development and tool building support.

O5. Emphasis is given to secondary requirements: Another interesting observation is that quality attributes that have been most frequently encountered in past work experiences (*e.g.*, *Software update*, *accessibility*) generally do not have a significant impact on the outcome of an industrial project (See "familiarity" and "importance" columns in Table 2). Further studies should analyze current design and specification methodologies and propose improvements on existing documentation practices.

Constraint	Tool	Reported Testing Tools
authorization	15%	SoapUI / *other:* Framework (JAAS)
throughput	26%	Meter, LISA, Selenium, Lucust, Gatling, HP LoadRunner
response-time	17%	JMeter, LISA, Selenium
data retention policy	8%	*no tool specified*
authentication	3%	*other:* Framework (JAAS, Spring)
data integrity	8%	Moose / *other:* db-constraints, Framework
visual design	4%	*other:* Framework
code quality	39%	Sonar, Findbugs, Code critics, Checkstyle, Emma, Clover
meta-annotation	19%	dclcheck
accessibility	0%	*no tool specified*
communication	8%	Moose, dclcheck
availability	10%	DynaTrace, Gomez, Shell script + Selenium, Pingdom
event handling	12%	dclcheck, Moose
data structure	16%	Moose / *other:* Custom tools
software infrastr.	8%	*other:* Automated declarative provisioning
signature	7%	Moose, JMeter, soapUI
dependencies	22%	SAVE, dclcheck, Patternity, Jdepend, Ndepend, Macker, IntensiVE, SmallLint, DSM tool
recoverability	0%	*no tool specified*
software update	0%	*no tool specified*
hardware infrastr.	6%	*no tool specified*
file location	0%	*other:* Guaranteed by framework
naming conventions	11%	Code critics, Checkstyle, PMD, FxCop, IntensiVE, Petit-Parser

Table 3. Survey results related to tool-aided architectural constraints testing. Columns (from left to right): Constraint name; respondents using third-party tools for testing the constraint; adopted tools.

O6. Tools do not take advantage of existing formalizations: Figure 1 shows that some constraints (*e.g.*, *dependencies*, *naming conventions*) are more often formally specified than automatically validated. However, formally specifying constraints without automatically verifying them is less than optimal. Based on our analysis, we observe that some adopted notations do not provide sufficient details to support validation (e.g. UML for describing *signature*) and other notations are not fully taken advantage of by the existing tools (e.g. regular expressions for describing *naming conventions*). We think that more empirical studies are needed in order to expose actual formalization practices. The results of these studies might expose common flaws of existing notations and provide concrete evidence of practitioner's needs.

5 Discussion

In this section we discuss some general strategies that could help address the issues raised in the previous section.

Reduce the Gap between Specification and Implementation As observed, many of the current tools force the user into a needlessly technical exercise. Several dependency testing tools (*e.g.*, JDepend, Dependometer), for example, not only require the test specification to be written using a technical notation (*i.e.*, Java or XML), but also offer poor documentation on how to do so.

Architects should be able to express their concerns in a single uniform format. Respondent **G** said that having the option to embed a formal (yet readable) test specification of his architectural rules in a Word document would be extremely appealing to him. This would allow him to write well-formed testing rules in a familiar environment with the additional benefit of automatic validation.

Terra *et al.* [20] and Marinescu *et al.* [15] proposed two different DSLs (Domain Specific Languages) for expressing quality requirements (See section 7). Both languages serve the purpose of encoding valuable information in a testable yet readable format. Unfortunately the expressiveness of such DSLs is strongly defined by the capabilities of the underlying tool. Völter [21] reports on a case study where a DSL is defined progressively by interacting with the customer. The language, grammar and support tooling is developed iteratively and will eventually be used as the basis for code generation and analysis. Cucumber[6], a behavior-driven development framework, is based on a similar concept. Tests are written by non-technical stakeholders and are checked by building an interpreter that translates the text into actual unit tests.

These approaches show that having business-readable descriptions of relevant design properties helps keeping alive the conversation between all involved stakeholders. It also shows that a well engineered DSL is useful for encoding information in a uniform and unambiguous manner, which can turn useful for supporting more sophisticated testing activities. We believe that users should not be asked to describe their quality requirements within the boundaries defined by a testing tool. Instead, tools should be employed to verify user-defined rules on a best effort basis.

Increase Awareness through Continuous Feedback Several respondents (**G**, **H**, **J**) use Sonarqube as a guide for driving code review activities. Sonarqube aggregates code analysis reports from multiple sources and presents them in a customizable web-based interface. Information is constantly kept up-to-date, well integrated and easy to navigate. All aspects exposed by the tool relate to general low-level characteristics of the system that are typically of little interest for architects. The strength of Sonarqube mostly seems to be bound to its integrability (analysis can be configured to run as a build step in a wide range of

[6]`http://cukes.info`

continuous integration servers), the concreteness of its result and the fact that all information are current and kept up-to-date.

Having seamless access to a comprehensive set of system-wide properties and infringed rules is a good way to exercise control over non-functional aspects of an implementation. If architects had the chance to define domain-specific rules for testing design constraints that are relevant for their architecture, they would be able to reach a higher and more targeted level of control. Our intuition is that monitoring platforms, such as Sonarqube would largely benefit from being integrated with highly customizable DSL-based tools (*e.g.*, DCL[20], InCode.Rules[15]). Being able to specify similar and more articulated rules on this and other aspects of the system would eventually reduce the generality of the results minimizing the number of false warnings and optimizing review-time.

6 Threats to Validity

Internal Validity During our first study, we tried to gather impressions and opinions by conducting semi-structured interviews. Our goal was to gather a clear answer to all the research questions presented in the introduction. All discussions have therefore been partially moderated by the interviewer. We did our best to minimize the influence of the interviewer on the respondent, but we cannot exclude the existence of biased answers. Some observations or questions made by the interviewer might have induced the respondent to articulate his answer in an unnatural way. The effect of a similar threat should have been mitigated by the number of different answers to the same question.

Users taking part in the survey had the right to remain anonymous. 41% of them chose not to share any identifying personal information (*i.e.*, email address). Among those, 71% (29% of the total population) did not specify their professional title. Due to this lack of information, we are unable to make general statements over the population participating to the survey. It would anyway be reasonable to assume that most of the people were either architects or professionals playing a comparable role. The fact that we contacted people belonging to our industrial collaborators network and that we posted invitations only on architecture-oriented virtual communities should support our hypothesis.

External Validity Another limitation could be seen in the relatively modest number of participants who participated in each phase of the study. The first study involved 14 respondents, while the survey counted 34 valid results. These numbers could appear small, but in fact are comparable to those reported by similar studies. Four out of five of all the interview-based studies centered around non-functional requirements [1] involve 14 or fewer participants. If we consider the surveys related to the same topic [1], we see that two out of four studies draw their conclusions based on fewer than 34 responses.

7 Related Work

In our work we discuss the nature of quality requirements and report on the techniques used for their verification. We examine both topics from a very pragmatic point of view, taking in consideration concrete examples and specific information. To the best of our knowledge, no other empirical study covers the same topics adopting a similar standpoint.

Several surveys related to NFRs (non-functional requirements) have been carried out (See related work by Ameller *et al.* [1]). The main outcome of all these studies often consists of a ranking showing how non-functional requirements compare based on the level of importance attributed by the users. All these studies focus on generic quality characteristics ignoring actual quality attributes that practitioners address in the requirements. Our study provides new insights from a complementary point of view, showing which quality attributes are considered relevant and providing details of their validation.

Poort *et al.* [18] found a statistical correlation between the verification of NFRs and project success. According to their results, the benefits of verification are also more significant if NFRs are verified in early stages of a project. In our study we explore how NFRs get actually validated in practice.

Various research contributions show that architecture-related requirements can be formalized using ADLs (architectural description languages). ADLs allow to model an architecture as a set of interlinked components enriched with a predefined meta-annotations. These models are typically weakly related with the implementation. Tools are sometimes provided for checking the semantic consistency of relationships and annotations but only at the model level. Moreover, there is scarce evidence that the general concepts defined in ADLs (*i.e.*, Components, Ports, *etc.*) actually reflect the the way architects think about their architecture. Case studies, showing evidence of the practical utility of the language, can only be found for a few of the most prominent ADLs (*i.e.*, AADL [6,5] and xADL [2]). We think that the lack of support for testing concrete architectures combined with the possible mismatch between offered features and real needs can be the cause of the — by now confirmed [14] — failure of adoption of ADLs by the general public. In this paper we draw observations that could help making ADLs more effective and useful.

Recent research efforts try to make up for these limitations by proposing more test-oriented ADLs. Terra *et al.* [20] proposed a specification language for expressing restrictions on the existence of certain types of relationships (*e.g.*, access, extension) between sets of classes. Marinescu *et al.* [15] supports the specification of undesired dependencies and class-level anti-patterns. Both ADLs are supported by custom-built testing tools that enable rule verification at the code level. Other languages (*i.e.*, SOUL [16] and LePUS3/Class-Z [9]) are more formal and support more complex specifications. They provide the means to validate quality requirements at code level, but also require considerable training before usage.

8 Conclusion

We presented the results of two empirical studies that explore how quality requirements are defined and validated in practice. The studies show that architects care about the validation of quality requirement but are often unable to make best use of the currently available tools.

We observe that the present offering of tools is limited in number and that several solutions are not able to satisfy common requirements (see section 5). Practitioners are rarely willing to develop solutions for governing architectural decay and are not motivated to formalize their quality requirements. Current formalization notations are typically strongly tied to specific testing solutions and are often lacking in readability. To improve this situation, we propose some ideas for specifying quality requirements and for reducing the cost of validation. Future testing solutions should take advantage of existing formalizations and provide functionalities that fulfill empirically recognized requirements.

In the future we plan to apply some of the discussed ideas by experimenting with new solutions for supporting the specification and validation of quality requirements.

Acknowledgment

We thank Erwann Wernli for valuable discussions regarding the content of this paper, and we thank the anonymous reviewers and the shepherd assigned to this paper for their many helpful suggestions. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We also thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

References

1. D. Ameller, C. Ayala, J. Cabot, and X. Franch. How do software architects consider non-functional requirements: An exploratory study. In *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pages 41 –50, Sept. 2012.
2. N. Boucké, A. Garcia, and T. Holvoet. Composing structural views in xADL. In *Proceedings of the 10th international conference on Early aspects: current challenges and future directions*, pages 115–138, Berlin, Heidelberg, 2007. Springer-Verlag.
3. S. J. Carrière and R. Kazman. The perils of reconstructing architectures. In *Proceedings of the third international workshop on Software architecture*, ISAW '98, pages 13–16, New York, NY, USA, 1998. ACM.
4. J. W. Creswell and Vicki. *Designing and Conducting Mixed Methods Research.* Sage Publications, Inc, 1 edition, Aug. 2006.
5. P. Feiler, D. Gluch, J. Hudak, and B. Lewis. Embedded systems architecture analysis using SAE AADL. Technical Report CMU/SEI-2004-TN-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2004.

6. P. Feiler, D. Gluch, and K. Woodham. Case study: Model-based analysis of the mission data system reference architecture. Technical Report CMU/SEI-2010-TR-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2010.
7. P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, DTIC Document, 2006.
8. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, New York, NY, USA, 2000.
9. E. Gasparis, J. Nicholson, and A. Eden. Lepus3: An object-oriented design description language. In G. Stapleton, J. Howse, and J. Lee, editors, *Diagrammatic Representation and Inference*, volume 5223 of *Lecture Notes in Computer Science*, pages 364–367. Springer Berlin Heidelberg, 2008.
10. M. Haigh. Software quality, non-functional software requirements and it-business alignment. *Software Quality Control*, 18(3):361–385, Sept. 2010.
11. ISO/IEC. ISO/IEC 25010 — Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, 2010.
12. P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.
13. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *ESEC'95: Proceedings of the 5th European Software Engineering Conference*, volume 989 of *LNCS*, pages 137–153, Sitges, Spain, Sept. 1995. Springer-Verlag.
14. I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What industry needs from architectural languages: A survey. *Software Engineering, IEEE Transactions on*, 39(6):869–891, 2013.
15. R. Marinescu and G. Ganea. inCode.Rules: An agile approach for defining and checking architectural constraints. In *Intelligent Computer Communication and Processing (ICCP), 2010 IEEE International Conference on*, pages 305–312, Aug. 2010.
16. K. Mens and A. Kellens. IntensiVE, a toolsuite for documenting and checking structural source-code regularities. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp. –248, mar 2006.
17. M. B. Miles and M. Huberman. *Qualitative Data Analysis: An Expanded Sourcebook(2nd Edition).* Sage Publications, Inc, 2nd edition, 1994.
18. E. Poort, N. Martens, I. Weerd, and H. Vliet. How architects see non-functional requirements: Beware of modifiability. In B. Regnell and D. Damian, editors, *Requirements Engineering: Foundation for Software Quality*, volume 7195 of *Lecture Notes in Computer Science*, pages 37–51. Springer Berlin Heidelberg, 2012.
19. R. Svensson, T. Gorschek, B. Regnell, R. Torkar, A. Shahrokni, and R. Feldt. Quality requirements in industrial practice — an extended interview study at eleven companies. *Software Engineering, IEEE Transactions on*, 38(4):923–935, 2012.
20. R. Terra and M. T. Valente. A dependency constraint language to manage object-oriented software architectures. *Softw. Pract. Exper.*, 39(12):1073–1094, Aug. 2009.
21. M. Voelter. Architecture as language: A story. *InfoQ*, Feb. 2008.